Wiring up a web page to a backend database can be tedious. You have to use some type of data access layer (DAL) and then map this to fields on a page. This is repetitive and error prone code. There are numerous ORM tools that allow you to interface with a database but this does not help you with loading controls, prompts, and validators on a page. This article will explore using nHydrate as a DAL that provides copious amounts of metadata and thus provides a release from some of the tedium of manually wiring up a web page.

First a short explanation of nHydrate, this is a platform that allows you develop software in a model driven way (or domain driven design, DDD). You design a model that contains entities and relations. There are generators that read the model and create code based on that model. When you need to make changes to your application design, simply change your model and re-generate. A database installer controls database changes and versioning. You never touch the database. The database is not the model. You have a real model. It really does cut down on the errors we all make when performing repetitive tasks.

The Entity Framework generated DAL provides metadata that can be used to automate many of the tasks that we perform on a web page. We will see how to create prompts and validators, as well as bind data to controls in a compile-time checked manner. Using the compiler is a big plus. I always try to put as much functionality in as possible. If you add code in markup via script or other mechanism, there will be issues when you change your data model or perform a host of other activities. Code is king; the compiler checks it!

This sample depends on a little code I have written around a model. A nHydrate model is used to create a DAL that interacts with the database. The generated Entity Framework layer has additional extensions, attributes, and metadata around it to facilitate writing applications quickly. My “ORMMap” class performs all of the generic functionality I need to make a web page work. This class is not part of the generated framework as it is very specific to my implementation. How you choose to bind or map a user interface is specific, based on your technology such as web forms, MVC, WinForms, etc. You will need to write a little connection code like I have done. However the metadata around the DAL makes this easy.

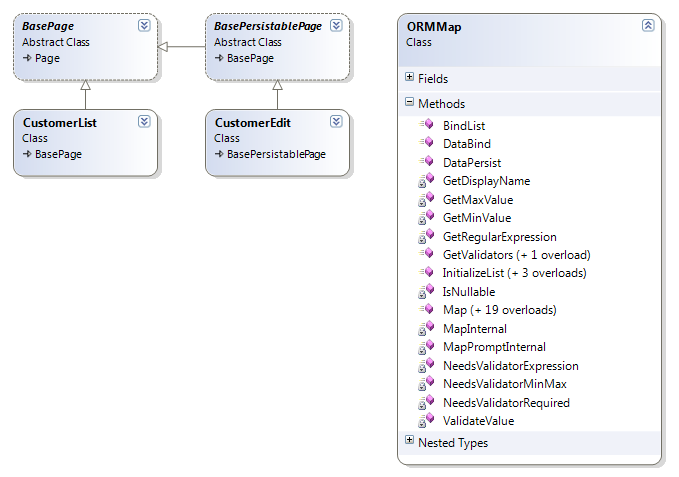
What this sample does:

* Dynamically inserts validators on page based on model metadata
  + Required validators
  + Range validators
  + Regular expression validators
* Verifies data types like integers in textboxes
* Sets maxsize for textboxes based on metadata
* Loads dropdown lists from database
* Loads paginated grids
* Handles paging of data
* Binds many kinds of UI controls to Entity Framework objects
* Validates saves and displays errors
* Eliminates viewstate

In this sample, I can bind labels, textboxes, check boxes, radio buttons, lists, and dropdowns. I can populate dropdowns, check lists and radio lists as well. There is very little code on my primary forms and the screens still have load/save functionality with validation.

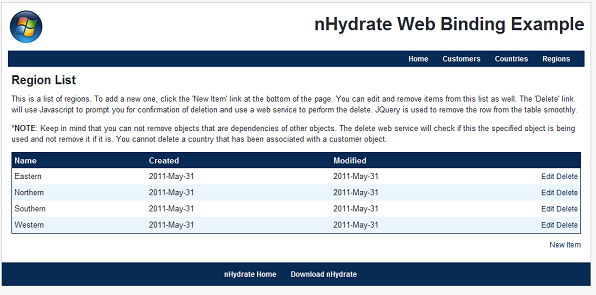
The connection code I wrote has only a few methods. The first allows you to load a combo box. In the page initialize I have added the list load code. This has the added benefit that I do not need viewstate. In fact all of the pages in the sample have viewstate turned off. The way we are binding here does not require viewstate at all. Let us look at an add/edit page for a "Region" object first. Our model has 3 entities: Customer, Country, and Region. A Customer has [0..1] Countries and [0..1] Regions. This is a very simple model used to demonstrate how to can bind objects with associations.

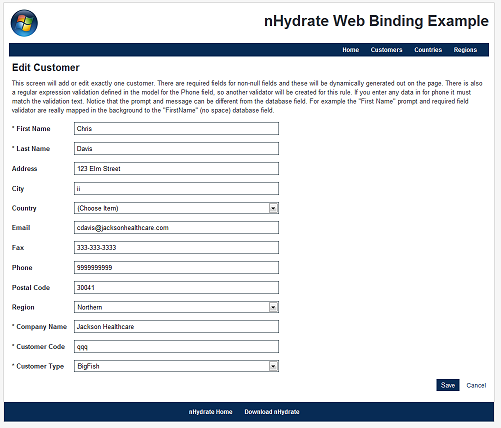
My project has 2 base pages from which I derive all pages. The first is BasePage which does nothing. It is merely a base. The second is BasePersistablePage which is used for edit pages and has core functionality for loading and saving. It has three main methods: SetupBindings, CreateObject, and SaveData. An edit page overrides these methods to add any custom code needed to the bind. Below is a class diagram of my main site objects. You can see an edit and list page along with their base types. The “ORMMap” object is expanded to show its methods. In this class there is no code that deals with specific objects (list Region or Customer). This class can be used from any page to handle any generated object.



Keep in mind that these pages are not part of the generated framework. There is no reason to use my syntax or code. I have created the “ORMMap” class and the base pages to abstract out the functionality of loading and saving nHydrate objects. I mention this because it might be a strange syntax for you and I have seen enough code generators to know they use weird and cumbersome syntax that confuses me. nHydrate is straight Entity Framework. If you have used EF then there is nothing foreign in the objects I am using in code.

Below are two screen shots the first of a list if Region objects. The second is an edit page of a Customer. I have included the customer edit screen because it is the most complicated and contains dependent objects. The screen is quite simple in that it is just a list of controls that allows for data entry.





Let us start with the Region object because it is very simple. It has an ID and a Name property. This is the classic look-up table. Databases are full of these for states, countries, regions, etc. They have just a primary key and a name. When a user edits one of these objects, he is really only editing the name as the primary is never shown or edited. In the region edit page, I override the “SetupBindings” method. The database context is passed to me so I use it to look up the actual object I wish to bind. I am pulling the primary key off of the URL string. This sample website uses a URL structure like "/regionedit.aspx?id=2". This makes it easy for users to bookmark a page. In the code, I also change the header to indicate to the user whether this is a create or edit operation.

Please notice the last line of code. This will bind the textbox named "txtName" to the region object. All generated entities have associated enumerations created that map to their fields. This allows you write methods that can work on specific fields via enumeration. These are always in sync because they are generated from your nHydrate model, just like the objects themselves. This line of code is mapping the “Name” field of a Region object to the textbox. This one line of code will two-way bind the EF object to the UI control. There are no magic strings here. I did not put the field designation “Name” in quotes (like a dataset). I used the provided enumeration for the Region object.

There is only one line of code to map the textbox since there is only one textbox. You will need one line of code for each control. There are numerous overloads, one for each control type of course. You could add more complicated overloads for other .NET controls or even third party controls you buy. Simply add an overload for a new control type and add whatever logic is necessary to handle the new control type.

|  |
| --- |
| protected override void SetupBindings(EFExampleEntities context) {  int id = this.Request.GetURLInteger("id");  var item = context.Region.Where(x => x.RegionId == id).FirstOrDefault();  lblHeader.Text = "Edit Region";  if (item == null)  {  lblHeader.Text = "New Region";  item = CreateObject(context) as Region;  }    this.Mapper.Map(txtName, item, Acme.EFExample.EFDAL.Entity.Region.FieldNameConstants.Name); } |

There is not much more code on the page than that. There is a “CreateObject” method on the page that is called when the base page needs an object created, but this just creates a new object and adds it to the context by default. If you have more complex business rules to perform when adding this object type you could add it or call that logic from here.

My entire Region edit page is about 50 lines of code. I would argue that this is really small. Keep in mind that any necessary validation that is defined in the model would automatically be added.

The Customer edit page is a bit more complex of course but has the same structure. We must load additional controls so it makes sense to look at its code too. In this code we see the “InitializeList” method used. This is called to load a dropdown with the list of values from which the user can choose one. We simply pass in the control, the list to bind and the fields that will be used for text and value. These are all loaded in the OnInit event so there is no need for viewstate in my example. Later in the code we bind each control individually to the appropriate property on a specific object. In this sample all fields bind to the same object; however this is not necessary. We could just as easily bind some of the UI controls to dependent objects off the primary or any other EF object we wish.

|  |
| --- |
| protected override void SetupBindings(EFExampleEntities context)  {  int id = this.Request.GetURLInteger("id");  var item = context.Customer.Where(x => x.UserId == id).FirstOrDefault();  lblHeader.Text = "Edit Customer";  if (item == null)  {  lblHeader.Text = "New Customer";  item = CreateObject(context) as Customer;  }    //Load the dropdowns  this.Mapper.InitializeList(cboCountry, context.Country.OrderBy(x => x.Name).ToList(), Acme.EFExample.EFDAL.Entity.Country.FieldNameConstants.Name.ToString(), Acme.EFExample.EFDAL.Entity.Country.FieldNameConstants.CountryId.ToString());  this.Mapper.InitializeList(cboCustomerType, context.CustomerType.OrderBy(x => x.Name).ToList(), Acme.EFExample.EFDAL.Entity.CustomerType.FieldNameConstants.Name.ToString(), Acme.EFExample.EFDAL.Entity.CustomerType.FieldNameConstants.CustomerTypeId.ToString());  this.Mapper.InitializeList(cboRegion, context.Region.OrderBy(x => x.Name).ToList(), Acme.EFExample.EFDAL.Entity.Region.FieldNameConstants.Name.ToString(), Acme.EFExample.EFDAL.Entity.Region.FieldNameConstants.RegionId.ToString());    //Bind the fields  this.Mapper.Map(txtFirstName, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.FirstName, lblFirstName);  this.Mapper.Map(txtLastName, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.LastName, lblLastName);  this.Mapper.Map(txtCity, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.City, lblCity);  this.Mapper.Map(txtAddress, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.Address, lblAddress);  this.Mapper.Map(txtCode, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.Code, lblCode);  this.Mapper.Map(txtCompany, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.CompanyName, lblCompany);  this.Mapper.Map(txtEmail, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.Email, lblEmail);  this.Mapper.Map(txtFax, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.Fax, lblFax);  this.Mapper.Map(txtPhone, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.Phone, lblPhone);  this.Mapper.Map(txtPostalCode, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.PostalCode, lblPostalCode);  this.Mapper.Map(cboCountry, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.CountryId, lblCountry);  this.Mapper.Map(cboCustomerType, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.CustomerTypeId, lblCustomerType);  this.Mapper.Map(cboRegion, item, Acme.EFExample.EFDAL.Entity.Customer.FieldNameConstants.RegionId, lblRegion);  } |

Now let's look at a list page. The region list page shows a paginated list of regions from which you can choose to edit or delete items. One of the big issues with displaying lists to pagination. There is no need to worry about this anymore as nHydrate removes most of the drudgery from you. The following code pulls the page offset and records per page from the URL string and uses it to pull a list of data from the database. I am using a PagingURL class I have written to parse the URL into the segments I care about. The paging object holds the page offset and records per page and passes this information into the data query method. When this call returns there is additional property on the paging object named "TotalRecords" that will contain a count of all records that match the where condition, thus allowing you to build a pager control on screen with current page, total pages, records per page, and total record count.

Once the page of data is returned, I simply bind it directly to a grid. I have built a paging control that I use on list pages and it simply takes the returned paging information and builds a stylized pager. This is all that is really needed to build a list/edit data entry website.

|  |
| --- |
| using (var context = new EFExampleEntities()) {  var url = new PagingURL(this.Request.Url.AbsoluteUri);  var paging = new Widgetsphere.EFCore.DataAccess.Paging(url.PageOffset, url.RecordsPerPage);  grdItem.DataSource = context.Region.GetPagedResults(x => x.Name, paging);  grdItem.DataBind();  PagingControl1.Populate(paging); } |

Now let's look at how we can pull the metadata out of the generated objects. The “ORMMap” class is really nothing more than a list of controls that we associate with a field on an Entity Framework object. Each nHydrate Entity Framework object has an associated metadata class that can be pulled of via attributes and used to pull model information at runtime. Common metadata would be Display Name, Regular Expression, and Required.

The display name is a friendly name you define for a field. So for example "FirstName" in the database might be "First Name" or “PostalCode” might be “Zip code”. This can be used for prompts and validation controls. To pull the display name from a generated object you can use the following code. The entity passed in is the base nHydrate EF object. All generated objects are derived from this type, so this routine will work on any of them.

|  |
| --- |
| private string GetDisplayName(Enum field)  {  var context = new EFExampleEntities();  var metadata = context.GetMetaData(context.GetEntityFromField(field));  var a = metadata.GetType().GetField(field.ToString()).GetCustomAttributes(typeof(System.ComponentModel.DataAnnotations.DisplayAttribute), true).FirstOrDefault();  if (a == null) return field.ToString();  else return ((System.ComponentModel.DataAnnotations.DisplayAttribute)a).Name;  } |

If there was not display name defined in the model, the database name is used. You can do the same thing to pull off the validation, regular expression and the required (non-nullable) attribute.

|  |
| --- |
| private bool IsNullable(Enum field)  {  var context = new EFExampleEntities();  var metadata = context.GetMetaData(context.GetEntityFromField(field));  var a = metadata.GetType().GetField(field.ToString()).GetCustomAttributes(typeof(System.ComponentModel.DataAnnotations.RequiredAttribute), true).FirstOrDefault();  return (a != null);  }    private string GetRegularExpression(Enum field)  {  var context = new EFExampleEntities();  var metadata = context.GetMetaData(context.GetEntityFromField(field));  var a = metadata.GetType().GetField(field.ToString()).GetCustomAttributes(typeof(System.ComponentModel.DataAnnotations.RegularExpressionAttribute), true).FirstOrDefault();  if (a == null) return string.Empty;  else return ((System.ComponentModel.DataAnnotations.RegularExpressionAttribute)a).Pattern;  } |

Notice the pattern is simply to get the metadata class of an object which can be retrieved via an extension method on all generated objects. The metadata class has all fields for the tracked object on it with attributes defining these data points like required, expression, min-max ranges, etc. You can use the descriptor class to build complex UI structures with non-specific code that works for all of your entities for any model. You could abstract this out to a common assembly that you use for all projects.

Another important and time-saving function point to use is to emit out validators. A common issue with UI development is knows which fields should have validation on them and keeping up with it as the DBA changes the database or project managers change business rules. It is easy to forget to change some arbitrary screen in the midst of all other development especially since these are business rules and not errors. Now a simple and elegant solution is at hand. Simply use the metadata to emit the validations on the page. If someone changes the nHydrate model and re-generates, well the validators are now model-driven so they will reconfigure themselves with no code changes.

The following routine returns a list of validators that you can add to the controls collection of a form. Simply call this method to get a list to dynamically add to your page.

|  |
| --- |
| public virtual IEnumerable<System.Web.UI.Control> GetValidators() {  var retval = new List<System.Web.UI.Control>();  foreach (var control in \_controlList.Keys)  {  var e = \_controlList[control];    //Determine if we need a required field validator  if (NeedsValidatorRequired(e.Entity, e.Field, e.CanHaveRequiredValidation, e.ForceRequiredValidation))  {  var r1 = new RequiredFieldValidator  {  ControlToValidate = control.UniqueID,  Display = ValidatorDisplay.None,  ErrorMessage = "The '" + GetDisplayName(e.Entity, e.Field) + "' is required!"  };  retval.Add(r1);  }    //Determine if we need a regular expresion validator  if (NeedsValidatorExpression(e.Entity, e.Field))  {  var r1 = new RegularExpressionValidator  {  ControlToValidate = control.UniqueID,  Display = ValidatorDisplay.None,  ErrorMessage = "The '" + GetDisplayName(e.Entity, e.Field) + "' is not the correct format!",  ValidationExpression = GetRegularExpression(e.Entity, e.Field)  };  retval.Add(r1);  }    }  return retval; } |

Notice that it checks to determine if a field is required and if so emits a required validator. It then determines if the Entity.Field has a validation expression and if so emits a regular expression validator. You can make this as specific to your situation as necessary. These were just two very easy ones to add. Also keep in mind that this is all generic code. This does not work on any one specific object like Country or Customer. It works on all objects in any model you can build.

This simple example demonstrates how you can create quite robust data entry screens with very little code. The whole idea is to use metadata to drive your application. This is much easier with model driven development. If your model supports extra metadata properties and is not just an ORM-mapper you can handle quite complex scenarios quite generically.

This will get you started with model driven development of a web UI. This technique allows you to keep your markup small. We did not add validators manually and I did not even mention that the MaxLength property of text boxes are set. This ensures that users cannot enter data that is too big for the field. Information that can be defined in metadata can be emitted out as code and mark-up at runtime, thus reducing the code you write and the associated mistakes.